# A Domain Specific Language for Expressing Model Matching

Kelly Garcés, Frédéric Jouault, Pierre Cointe, Jean Bézivin

# A Domain Specific Language for Expressing Model Matching

**Kelly Garcés [*] [§] [£] — Frédéric Jouault [+] [§] — Pierre Cointe [*] [£] — Jean Bézivin [+] [§]**

[*] *École des Mines de Nantes*
[+] *INRIA, Centre Rennes - Bretagne Atlantique*
[§] *AtlanMod team*
[£] *ASCOLA team, LINA (UMR 6241)*

*{kelly.garces, pierre.cointe}@emn.fr*
*{frederic.jouault, jean.bezivin}@inria.fr*

RÉSUMÉ. *Une stratégie de mise en correspondance calcule des liens entre deux modèles en exécutant un ensemble d'heuristiques. Dans ce papier, nous introduisons AML (pour* AtlanMod Matching Language*), un langage dédié à l'expression des stratégies de mise en correspondance des modèles. AML est conçu pour l'ingénierie des modèles et il implémente les stratégies de mise en correspondance par des chaînes de transformation de modèles. Chacune de ces transformations prend un ensemble de modèles en entrée et crée un modèle de correspondances en sortie. Nous présentons un compilateur qui prend en entrée des programmes AML et génère en sortie du code ATL (AtlanMod Transformation Language) et Apache Ant. Le code ATL instrumente les transformations de modèles de mise en correspondance et le code Ant orchestre leur exécution. Nous évaluons cette implantation sur deux stratégies de mise en correspondance composé d'ensembles de transformations issues de la litérature.*

ABSTRACT. *A matching strategy computes mappings between two models by executing a set of heuristics. In this paper, we introduce the AtlanMod Matching Language (AML), a Domain Specific Language (DSL) for expressing matching strategies. AML is based on the Model-Driven paradigm, i.e., it implements model matching strategies as chains of model transformations. A matching model transformation takes a set of models as input, and yields a mapping model as output. We present a compiler that takes AML programs and generates ATL (AtlanMod Transformation Language) and Apache Ant code. The ATL code instruments the matching model transformations, and the Ant code orchestrates their execution. We evaluate this implementation on two strategies including robust matching transformations from the literature.*

MOTS-CLÉS : *Génie des Modèles, Transformation de Modèles, Mise en correspondance.*

KEYWORDS: *Model-Driven Engineering, Model Transformation, Matching.*

## 1. Introduction

Business and technological changes influence the evolution and integration of software systems. Software engineers have to deal with the heterogeneity among all the artifacts (e.g., many versions for each software artifact) in order to get the systems working. Various solutions have been proposed to tackle this issue. One of them, called *Matching*, has been thoroughly studied in the ontology and database schema contexts. The matching operation is also gaining importance in Model-Driven Engineering (MDE) (Falleri *et al.*, 2008).

An MDE system basically consists of a set of models, i.e., terminal models and metamodels (Jouault *et al.*, 2006a). A metamodel is composed of concepts and relationships. A terminal model contains instances of the metamodel concepts. In general, the matching operation computes mappings between two models. A mapping is a relationship between two model elements similar to each other in a way. The research problem of the following works may illustrate the need of mappings between terminal models and/or metamodels :

– **Metamodel-to-metamodel (M2-to-M2).** (Cicchetti *et al.*, 2008) describes the problem of adapting terminal models to their metamodels evolving. They uses the mappings and differences between two metamodels (i.e., a given metamodel to a former version of the same metamodel) for deriving adaptation transformations.

– **Terminal model-to-terminal model (M1-to-M1).** (Javed *et al.*, 2008) recovers metamodels from a repository of terminal models. The approach translates the terminal models into grammar-based representations, an engine then infers a metamodel from them. This work may motivate the need of mappings between two terminal models from which we can obtain metamodels.

– **Terminal model-to-metamodel (M2-to-M1 or M1-to-M2).** (Favre, 2004) points out that software systems may rapidly become out of sync with the implementation. In MDE systems, an instance of this problem is that the developers often change code without updating the corresponding terminal models and metamodels. In this scenario, we imagine the mappings, between a terminal model (extracted from source code) and a given metamodel, as artifacts for reconstructing the metamodel (e.g., for finding out new metamodel types).

There are many strategies to deliver these kinds of mappings (we will refer to these strategies as *matching strategies*). They typically execute a set of matching heuristics. Inspired by them, we have proposed MDE matching strategies that find out M2-to-M2 mappings (Garcés *et al.*, 2008). Every strategy has been implemented as chains of model transformations, where each model transformation instruments a matching heuristic. A Matching Transformation (indicated as MT in the remaining sections) takes as input a set of models (the models to be match –$ModelA$ and $ModelB$–, a mapping model returned by other transformation, and additional models), and yields a mapping model conforming to a mapping metamodel.

After studying many matching strategies and performing some experimentations, (see full results in (Garcés *et al.*, 2008)), we have observed two main issues that impact the productivity of matching developers :

1) **Coupling of strategies and model representations.** Most of matching approaches (Ohst *et al.*, 2003)(Xing *et al.*, 2005)(Girschick, 2006)(Treude *et al.*, 2007) write heuristics in terms of one internal model representation (as it will have to be). Even though such heuristics compare very standard model features (e.g., labels, structure), these may be no longer applicable when the models to be matched conform to other metamodels. The solution often is to rewrite a given heuristic for each pair of metamodels. As a result, we get multiple heuristics encompassing the same matching logic.

2) **Lack of matching logic constructs.** Matching heuristics have been mostly instrumented using General-Purpose Languages (GPL), e.g., (Melnik *et al.*, 2003) and (Do, 2005). Matching experts may spend much more time for expressing a heuristic, using the GPL constructs, than the time for inventing it. Considering that the number of heuristics is rapidly increasing (OAEI, 2008), the question is : How can we reduce effort to express matching heuristics ?

By considering the fairly good results of our early MDE matching strategies (Garcés *et al.*, 2008), we have decided to make more generic the original approach (i.e., to develop not only strategies to match two metamodels, but to match any pair of models), and to tackle the two issues described above. To achieve this goal, we propose the AtlanMod Matching Language (AML), a DSL that provides constructs for straightforwardly expressing matching strategies. We deal with the coupling problem using a compilation strategy which adapts matching rules to the models to be matched. Whereas a detailed discussion about the compilation strategy is out of scope of the paper, this do focus on the language constructs, and the first significant results of using them.

An outline of the remainder of this paper follows. Section 2 explains the matching domain and the kinds of heuristics, and also introduces a motivating example. Section 3 describes the AML constructs and their semantics. Section 4 shows how to specify the motivating example using AML. Section 5 reports the experimentation results. Section 6 presents the related works. Finally, Section 7 concludes the paper.

## 2. Matching Domain Overview and Motivating Example

After analyzing the strategies compiled in (Euzenat *et al.*, 2007), we can characterize the matching domain. A matching strategy is a stepwise process that takes two models $A$ and $B$, along with optional domain knowledge, and produces mappings between the models. A mapping relates (links) an element of $A$ at most one element of $B$. A mapping has a similarity value (between 0 and 1) that represents how similar the linked elements are. A matching strategy deliveries mappings by applying at least 5 kind of heuristics :

– **Creation** establishes a mapping between the element $a$ (in model $A$) and the element $b$ (in model $B$) when these elements satisfy a condition.

– **Similarity** computes a similarity value for each mapping prepared by the creation heuristics. One function establishes the similarity values by comparing particular model aspects : labels, structures, and/or data instances (the latter has sense when the models represent metamodels). In general, a given comparison is direct and/or indirect. Direct means only the model elements, e.g., $a$ and $b$, are compared each other. The indirect comparison separately examines $a$ and $b$ in relation to a third element (e.g, $c$) from a domain knowledge resource (e.g., an ontology).

– **Aggregation** combines similarity values by means of one expression. An expression often involves :

- *n*, the number of heuristics providing mappings.
- $\delta(a_i, b_i)$ similarity value computed by the heuristic *i*
- $w_i$ weight or importance of the heuristic *i*, where $\sum_{i=1}^{n} w_i = 1$

– **Selection** selects mappings whose similarity values satisfy a condition.

– **Rewriting** retypes/reorganizes generic mappings that fill a condition. "Retypes" intends to transform a generic mapping to a richer semantic mapping. "Reorganizes" means to arrange mappings when there is a kind of relationships between the linked elements.

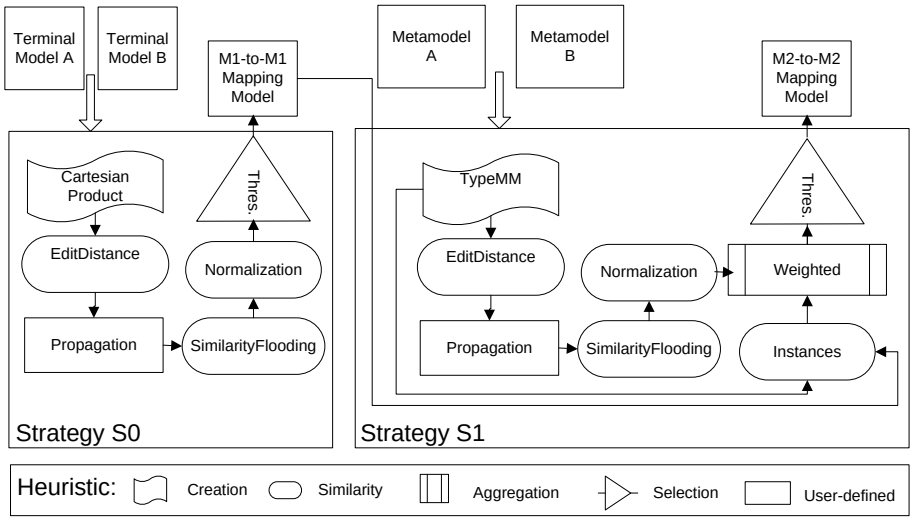


**Figure 1.** *The strategies S0 and S1*

Fig. 1 shows a motivating example whose purpose is to find out M2-to-M2 mappings using (among other model aspects) data instances. The main strategy,

i.e., S1, matches two metamodels $MetamodelA$ and $MetamodelB$, conforming to the same metametamodel, e.g., EMF/Ecore, KM3. In particular, S1 uses the matching transformation Instances which propagates similarity values of M1-to-M1 mappings to M2-to-M2 mappings. The M1-to-M1 mappings have been earlier computed by the strategy S0. The strategy S0 therefore matches two terminal models, $TerminalModelA$ and $TerminalModelB$, conforming to the metamodels $MetamodelA$ and $MetamodelB$. Remark on both S0 and S1 reuse the matching transformations EditDistance, Propagation, SimilarityFlooding, Normalization, and Threshold. The difference is that the transformations match terminal models in S0, and metamodels in S1. Note that Fig. 1 uses a particular symbol to indicate the type of each transformation. The figure incorporates an additional type, user-defined, which represents heuristics capturing functionality beyond our classification, e.g., Propagation. We describe these transformations in Section 4.

## 3. The AtlanMod Matching Language (AML) in a Nutshell

### 3.1. *Overview*

AML is a DSL for expressing MDE matching strategies. AML keeps the declarative flavor (of some model transformation languages) for expressing matching transformations, and adds dataflow programming constructs for describing matching transformation chains. Fig. 2 shows the tools that support AML :

**The Compiler** takes a given AML program (including the constructs matching rule and model flow) and performs the following tasks :

– Analyze the AML program (syntax and semantic)

– Merge pre-existent matching rules (which are available at a library of strategies)

– Generate a model matching transformation (written in a concrete model transformation language, i.e., ATL) for each matching rule. This transformation contains the matching rule, and additional copying rules. The ATL compiler generates executables from the code.

– Translate the model flow section into transformation chain specification code, i.e., Ant scripts.

**The Launcher** executes Ant scripts. This specifies the input models to the transformations, and saves the mapping models into a repository.

In this work we envision AML to specify matching strategies as automatic as possible. Model flow allows to specify no user assistance. The user can nonetheless refine mapping models when a strategy execution ends, and apply strategies on manually refined mapping models. The first AML version provides no constructs for rewriting matching logic (described in section 2). As this logic highly depends on a given domain (e.g., metamodel evolution), more work is needed to determinate whether

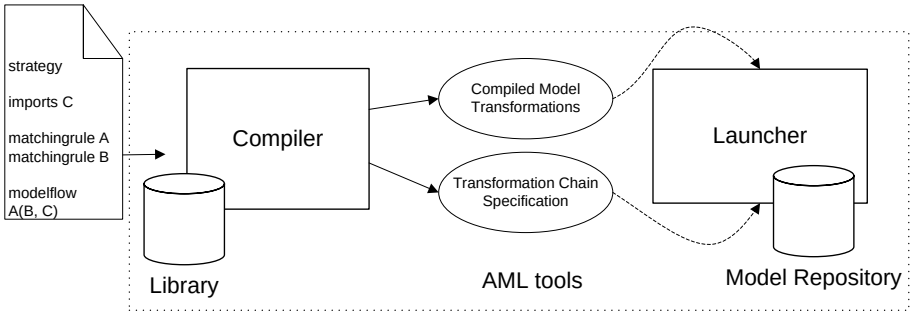


**Figure 2.** *AML tools*

constructs can factorize its patterns. Although there exist these limitations, AML remains applicable in matching domain.

### 3.2. *Concrete syntax and semantic*

Listing. 1 gives the overall structure of a matching strategy. A matching strategy needs a name, and contains an import section, a set of matching rule declarations, and a model flow block. Below we present the concrete syntax and informal semantics of the AML constructs.

Listing 1 – Overall structure of a matching strategy

```
strategy S1 {
  imports BasicStrategy;
  uses Propagation(mappingM : MappingMM);
  create TypeMM () {...}
  ...
  sel Threshold () {...}
  modelFlow {...}
}
```

**Import section** enables to declare AML strategies to be imported. The imported strategies contain matching rules supposed to be reused in a model flow block.

Listing 2 – Import section

```
strategyreflist := (imports strategyref)? ;
strategyref := name | name , strategyref
```

**Matching rule declaration** specifies the matching rule type, followed by a name, the list of input models (indicating their respective metamodels), the list of libraries to be imported, and the matching rule body.

Listing 3 – Matching rule declaration

```
matchingrule := (create | sim | aggr | sel | uses) name  inputmodellist
    ↪libraryreflist? { matchingrulebody }
```

```
inputmodellist := () | ( inputmodel )
inputmodel := model | model , inputmodel
libraryreflist := uses libraryref
libraryref := library | library , libraryref
```

The developer declares a matching rule using the keywords `create`, `sim`, `aggr`, `sel`, or `uses`. The keyword `uses` declares an external ATL transformation, which should be executed somewhere in the strategy. The developer should only specify the name and list of input models of the ATL transformation. The transformation functionality is therefore defined in a separated ATL module. It is not necessary to specify the mapping model (intended to be manipulated by the matching rule) in the input model list, this remains intrinsic. A library list enables to import ATL libraries containing helpers. A helper can be used to specify operations on model elements (Jouault *et al.*, 2005). In AML, every helper is declared in ATL libraries.

Matching rule body specifies the expressions `inpattern`, `variables`, and `simexp`. An inpattern indicates a set of source types coming from the metamodels specified in a rule input model list. Most of matching rules establish a condition, i.e., an OCL expression (OMG, 2006). When the condition is true, then the mappings are manipulated in some way. In the creation and similarity rules the condition involves the source types and the Mapping type. In the selection rules, in order, the condition involves the similarity value. There is no condition in the aggregation rules. The variables section makes possible to declare a number of local variables. The variables declared in this section can be used in sim expressions. Sim expressions specify similarity functions, these are mandatory in similarity and aggregation rules.

Listing 4 – Matching rule body

```
matchingrulebody:= inpattern? variables? simexp?
variables := using { variableslist }
variablelist := variabledecl | variabledecl ; variablelist
variabledecl := varname : vartype = initexp;
inpattern := intypelist condition
intypelist := (from intypedecl*)?
condition := (when oclexpression)?
intypedecl := varname : type in modelname ;
type := metamodelname ! typename
simexp := is oclexpression
```

**Model flow block** allows to declare how models flow among matching transformations. A model flow block consists of a set of model variable declarations.

Listing 5 – Model flow section, concrete syntax

```
modelflowblock := (modelFlow { modelvardecl* })?
modelvardecl := (modelname = )? modelflowexp
modelflowexp := (modelexp | transformationcall | weightmodelflowexp )
transformationcall := name parammodelflowexplist
parammodelflowexplist := () | parammodelflowexp
parammodelflowexp := modelflowexp | modelflowexp , parammodelflowexp
weightmodelflowexp := weight : transformationcall | modelexp
modelexp := model
model := modelname : metamodelname
```

A model variable declaration associates a model name to a model flow expression. This can be an input model, a transformation call or weighted model flow expression.

When the model flow expression is a `transformationcall(...)`, the AML system serializes the transformation output as a model conforming to the mapping metamodel.

A transformation call refers to a transformation rule by means of its name. While a transformation rule signature just contains input models, the transformation call indicates the input models and the mapping models to be manipulated. Observe that the developer can refer to input models or output models using a model flow expression or its corresponding model name (if the expression has been previously associated to a model name).

As a final point, weighted model flow expressions allow to associate a weight to a transformation call or to an input model expression (if this refers to a mapping model). An aggregation transformation call should use weighted model flow expressions as parameters.

## 4. The Motivating Example Using AML

This section presents how to specify the main blocks of a given AML strategy, i.e., model flows and matching transformations.

### 4.1. *Model flow of strategy S1*

Listing. 6 illustrates the model flow block of the strategy S1. We have omit the S0's model flow due to space constraints, we have taken S1 as a reference because it reuses S0's matching transformations, and it is a more elaborated strategy. The strategy S1 executes 8 MTs. Unlike the transformation TypeMM, the subsequent transformations take the previously produced mapping model as input. Listing. 6 distinguishes different model flow block features. For example, the block contains the two types of model variable declaration. While line 2 declares an additional input model, the rest of lines state matching transformation calls. In line 5, we directly use transformation call as arguments, instead of declaring the transformation call first, and then referring the output models. Whereas it is necessary to declare additional models and mapping models as transformation call parameters (for example, inst and typeMM in line 6), AML keeps the models to be matched intrinsic, e.g., $TerminalModelA$ and $TerminalModelB$ in S0, and $MetamodelA$ and $MetamodelB$ in S1. Finally, the model flow block contains an aggregation MT call (line 8). This notation enables to indicate the importance of similarity values returned by the MTs Normalization and Instances.

### 4.2. *Matching rules of strategy S1*

We discuss below the main MTs used in the strategies S0 and S1. Each matching transformation has an AML code listing whose number is enclosed between parenthe-

sis. In brief, the main differences between the AML matching transformations and the corresponding ATL versions are :

1) AML constructs hide source and target patterns that respectively specify : a) types of conformance of models to be matched, and b) mapping metamodel concepts. We can refer to them using the constructs thisLeft, thisRight, and thisMapping. The developer uses thisLeft to refer to elements of model $A$, thisRight to relate to elements $b$ of model $B$, and thisMapping to refer to mapping elements. In particular, the former constructs allow to develop matching rules without the matter of specifying the metamodel types to be matched. This makes some matching rules generic and applicable to any pair of models.

2) In the AML versions only remain source patterns that queries additional input models, conditions, and functions modifying similarity values.

3) AML provides constructors than factorize code, e.g., Summation, thisWeight, thisSim, thisInstances, thisModel.

Listing 6 – Model flow of strategy S1

```
modelFlow {
  inst = Instance:EqualMM;
  typeMM = TypeMM();
  levMM = EditDistance(typeMM);
  normMM = Normalization(SimilarityFlooding(Propagation(levMM)));
  instanceMM = Instances(typeMM, inst);
  weightedMM = Weighted(0.5:normMM, 0.5:instanceMM);
  thresMM = Threshold(weightedMM);
}
```

**TypeMM** (Listing. 7) creates a mapping for each pair of model elements having the same type. In contrast to TypeMM, the MT Cartesian Product (in S0) simply specifies true as condition. As a result, we get a mapping for each pair of model elements (without restrictions).

Listing 7 – TypeMM

```
create TypeMM () {
  when
    thisLeft.type = thisRight.type
}
```

**EditDistance** (Listing. 8) assigns a similarity value to each mapping prepared by the transformation CartesianProduct. The values are computed by the helper editDistance contained in the ATL library Strings. This compares the names of left and right model elements, and returns a value depending on the edition operations that should be applied to a name to obtain the other one. More edition operations, less similarity value.

Listing 8 – EditDistance

```
sim EditDistance uses Strings {
  is thisLeft.name.editDistance(thisRight.name)
}
```

**SimilarityFlooding** (Listing. 9) propagates previously computed similarity values. This is inspired on Melnik's algorithm (Melnik *et al.*, 2002). Our implementation has two steps. The first step creates an association (i.e., a PropagationEdge) for each pair of mappings (*m1* and *m2*) whose linked elements are related each other. For example, this step associates the mappings `(Class1, Class1')` and `(Attribute1, Attribute1')` because `Class1` of *A* contains `Attribute1`, and `Class1'` of *B* contains `Attribute1'`. The second step propagates a similarity value from *m1* to *m2* because of the relationship. As the first step requires not only mappings but also propagationEdges, we implement it using an external ATL transformation. SimilarityFlooding then takes the external transformation result, and propagates similarity values along mappings. Besides Similarity Flooding, the strategies S0 and S1 execute the MT Normalization which makes similarity values conform to the range [0,1].

Listing 9 – Similarity Flooding

```
1 sim SF () {
2   is
3         thisSim
4         +
5         MappingMM!PropagationEdge.allInstances()
6           ->select(e | e.incomingLink = thisEqual)
7           ->collect(e | e.propagation * e.outgoingLink.sim)
8           ->sum()
9 }
```

**Instances** (Listing. 10) propagates similarity values from M1-to-M1 mappings to M2-to-M2 mappings. The M1-to-M1 mappings are computed by the strategy S0. These mappings are supposed to be queried but not modified. We use the primitive thisInstances (line 3) to recover, for each M2-to-M2 mapping, e.g., linking the concepts $a$ and $b$, the M1-to-M1 mapping whose linked elements conforming to $a$ and $b$. The similarity function (line 5) calculates the average of the M1-to-M1 mapping similarity values. The function reuses the helper maxSim and introduces the helper totalSim (also included in the generic ATL library). This performs an addition of similarity values contained in a mapping set.

Listing 10 – Instances

```
1 sim Instances (instancesModel : MappingMM) {
2   using {
3     inst : Sequence(MappingMM!Mapping) = thisInstances(instancesModel);
4   }
5   is (maxSim(inst) * totalSim(inst)) / inst->size()
6 }
```

**Threshold** (Listing. 11) selects mappings with a similarity value higher than a given threshold. Line 2 specifies this condition.

Listing 11 – Threshold

```
1 sel Threshold () {
2   when thisSim > 0.7
3 }
```

**Weighted** (Listing. 12) computes a weighted sum of similarity values of mapping models. This needs relative weights associated to the models. In strategy S1, Weighted takes the mappings returned by the MTs Normalization and Instances, and their corresponding weights, i.e., 0.5–0.5 (see line 7 of Listing. 6). In Listing. 12, the expression Summation adds the similarity values (specified as thisSim), and multiplies them with weights (specified as thisWeight).

Listing 12 – Weighted

```
1 aggr Weighted () {
2   is Summation(thisSim * thisWeight)
3 }
```

## 5. Experimentation

This experimentation has three goals : 1) express robust matching strategies using AML, 2) apply matching transformations to metamodels and terminal models, and 3) evaluate the accuracy rendered by AML matching strategies.

### 5.1. *Prototype*

We have implemented the prototype on the AtlanMod Model Management Architecture (AMMA) platform (Kurtev *et al.*, 2006). More specifically, we have implemented the AML concrete syntax using the Textual Concrete Syntax (TCS) tool (Jouault *et al.*, 2006b). TCS generates a parser that converts an AML program from text to model format. We have developed the AML compiler by means of ATL code (of approximately 2300 lines). This code translates an AML model into : 1) a set of ATL matching transformations, and 2) an Ant file instrumenting the transformation execution chain. Additionally, we have used the AtlanMod Model Weaver (AMW) (Didonet Del Fabro *et al.*, 2005) to graphically manipulate the mappings models. We ran the benchmark on a PC with a 2.4 GHz Intel Core 2 Duo processor, 1GB of RAM, and Windows XP OS version 2002.

### 5.2. *Procedure and dataset*

We have implemented the strategies S0 and S1 using AML. The strategy S1 has been applied to the pair of metamodels Families and Persons (Eclipse.org, 2009), which respectively include the classes `Family-Member`, and `Person-Male-Female`. The strategy S0 in turn has been applied to terminal models conforming to these metamodels. The Families terminal model represents, for example, the family March. We have obtained the Persons terminal model by applying the transformation Families2Persons to the Families terminal model. The models have 8 elements on average. The selection of the models has been driven by three main reasons : 1) we can figure out the expected (correct) mappings from the transformation, 2) the mappings

**Tableau 1.** *Generated ATL code*

| Matching transformation | Number of lines |
|---|---|
| CartesianProduct | 90 |
| TypeMM | 90 |
| EditDistance | 75 |
| Instances | 80 |
| Normalization | 73 |
| SimilarityFlooding | 86 |
| Threshold | 75 |
| Weighted | 79 |

**Tableau 2.** *Accuracy measures*

| Measure | Value |
|---|---|
| Precision | 0.4 |
| Recall | 0.3 |
| Fscore | 0.38 |

are simple but interesting enough to be analyzed in a paper, and 3) the metamodels and terminal models are available as open-source.

### 5.3. *Results and discussion*

**Generated code.** Table. 1 shows the number of lines generated from each matching rule, involved in the strategies S0 and S1. The Ant Script contains 124 lines. The AML compiler generates this code in 4 seconds.

Although this study has been primarily applied on eight matching transformations and two strategies, it should be noted that AML considerably reduces the number of lines of code to be written. Comparing the number of lines of an AML transformation and its corresponding generated code, we find out that AML saves the codification of 80 lines of ATL code, and 10 lines of Ant code. We hope to implement more validations in future.

**Accuracy.** We have evaluated AML strategies accuracy using three metrics (Rijsbergen, 1979) : $Precision = \frac{CorrectFoundMappings}{TotalFoundMappings}$, $Recall = \frac{CorrectFoundMappings}{TotalCorrectMappings}$, and $Fscore = \frac{2*Recall*Precision}{Recall+Precision}$. The expected values of these metrics are between 0 and 1. The higher is the precision value, the smaller is the set of wrong mappings. The higher is the recall value, the smaller is the set of the mappings that have not been found. Fscore is a global measure of the matching quality. A high fscore value indicates a matching of high quality.

Table. 2 shows the accuracy values of S1, which have been influenced by the values of S0. In general, the values show that S1 has a relatively slow accuracy, matching the metamodels Families and Persons. Taking fscore as an example, the percentage of correct mappings is 38%. This value includes (for example) the correct mapping `(Member, Male)`. Based on these results, we have concluded that while the strategy S1 is robust enough, it is nevertheless inappropriate to find out mappings between the metamodels of our motivating example. The main reason is that the matching transformation Instances, which has an important percentage of confidence in S1 (50%), renders wrong matches. This is not associated to bugs in the code, but to the nature of instances-based heuristics, which render best results in a few particular cases (Doan *et al.*, 2001). The results thus suggest the need of a more suitable strategy to match the pair of metamodels Families and Persons. Unfortunately, we are unable to determinate from this data the correctness of the AML compiler. A way to do that may be to compare the accuracy of manually-developed transformations to transformations generated by the AML compiler. It seems a interesting work to do in further researchs.

## 6. Related works

There has been many previous work on the matching problem. In this section, we focus on the related works closer to MDE. We should nonetheless mention the works described in (Wang *et al.*, 2008)(Melnik *et al.*, 2002)(Do, 2005). They present robust strategies, that match schemas/ontologies, which have been implemented using GPLs. The strategies and reported lessons have inspired the conception of the AML constructs in some way.

In the MDE context, (Kolovos *et al.*, 2008) proposes a DSL, named Epsilon Comparison Language (ECL), for specifying model comparison algorithms. (Fleurey *et al.*, 2007)(Rubin *et al.*, 2008) in turn propose frameworks for model composition, i.e., matching and merging. We now compare these approaches and AML taking into account their contributions on model matching :

– Unlike the approaches described in (Fleurey *et al.*, 2007) and (Rubin *et al.*, 2008), AML consider not only homogeneous models but also heterogeneous, i.e., models conforming to metamodels different each other.

– Whereas ECL and (Fleurey *et al.*, 2007) offer only constructs to express similarity between two models to be matched (direct comparison). AML provides a richer set of concepts for expressing complete matching strategies. This includes indirect comparison strategies.

– (Fleurey *et al.*, 2007) and (Kolovos *et al.*, 2008) enable to express comparison algorithms that render binary values (0 or 1, equal or not equal). AML instead applies the notion of probabilistic mappings, which is closer to real world mappings.

– AML promotes matching heuristics not coupled to the models to be matched. The constructs thisLeft and thisRight allow to declare matching transformations without the matter of distinguishing metamodel types. This is improvement on (Fleurey

*et al.*, 2007)(Kolovos *et al.*, 2008)(Rubin *et al.*, 2008) which are metamodel type-based.

EMF Compare is an Eclipse.org tool for model comparison. Its matching strategies are hard-coded into the tool in Java. This prevents the customization of matching strategies necessary in many context, and supported by (Fleurey *et al.*, 2007), (Kolovos *et al.*, 2008), (Rubin *et al.*, 2008), and AML.

(Falleri *et al.*, 2008) automatically detect mappings between two metamodels using the algorithm *Similarity Flooding* described in (Melnik *et al.*, 2002). The algorithm is generic enough to match only metamodels. Remark on our AML version of the *Similarity Flooding* algorithm is applicable to metamodels and terminal models.

After having compared AML to other model comparison approaches, we compare its rule orchestration part to the Uniti (Vanhooff *et al.*, 2007) transformation chain definition framework. While Uniti is more declarative framework for transformation chain definition, AML constructs have been tailored for the representation of matching strategies in a more concise manner. Note that although the AML orchestration syntax is currently compiled into Ant tasks, it could also be compiled into Uniti. This design choice does not impact the user, and we simply chose the alternative we know better.

Although this section has concentrated on AML and the closer related languages, we should refer to (Steel *et al.*, 2007) as an approach dealing with the problem of coupling transformation and model representation. Instead of single-type based transformations, they propose a simple system based on a collection of interconnected types. Notwithstanding this approach augment the reuse of transformations, transformations remain brittle and restricted to the pre-defined type list. In contrast to (Steel *et al.*, 2007), AML leverages transformation reuse by means of constructs that hide model types at implementation time. AML compilation strategy therefore resolves the appropriate types at compilation time.

## 7. Conclusion

The main purpose of this paper has been to present AML, a DSL for expressing matching strategies. We have validated the DSL expressiveness by implementing two robust matching strategies, and eight matching transformations. The results are overall motivating. Firstly, we have developed matching transformation applicable to metamodels and terminal models. AML likewise saves the implementation of a significant amount of lines of code, i.e., 80 lines of ATL code, and 10 lines of Ant code (for each matching rule). The AML constructs moreover factorize code that make the strategies more readable. Thus, matching developers may analyze strategies, and modify them in order to improve mapping accuracy. From the results, it is also clear that more validations are needed, e.g., to implement more strategies, matching transformations, and to apply them to different pairs of models. Other further research may be : to compare the solutions to the problem of coupling between strategy and model representation, and to investigate constructs to express rewriting logic and matching user assistance.

## 8. Bibliographie


Cicchetti A., Ruscio D. D., Eramo R., Pierantonio A., « Automating Co-evolution in Model-Driven Engineering », *EDOC '08 : Proceedings of the 12th IEEE International EDOC Conference*, München, Germany, 2008.

Didonet Del Fabro M., Bézivin J., Jouault F., Breton E., Gueltas G., « AMW : A generic model weaver », *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*, 2005.

Do H. H., Schema Matching and Mapping-based Data Integration, PhD thesis, University of Leipzig, 2005.

Doan A., Domingos P., Halevy A., « Reconciling schemas of disparate data sources : A machine-learning approach », *In SIGMOD Conference*, p. 509-520, 2001.

Eclipse.org, *ATL Example Presentation, Families to Persons*, `http://www.eclipse.org/m2m/atl/basicExamples_Patterns/`. 2009.

Euzenat J., Shvaiko P., *Ontology Matching*, Springer, Heidelberg (DE), 2007.

Falleri J.-R., Huchard M., Lafourcade M., Nebut C., « Metamodel Matching for Automatic Model Transformation Generation », *in* K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, M. Völter (eds), *MoDELS*, vol. 5301 of *Lecture Notes in Computer Science*, Springer, p. 326-340, 2008.

Favre J.-M., « CacOphoNy : Metamodel-Driven Architecture Recovery », *WCRE*, IEEE Computer Society, p. 204-213, 2004.

Fleurey F., Baudry B., France R. B., Ghosh S., « A Generic Approach for Automatic Model Composition », *in* H. Giese (ed.), *MoDELS Workshops*, vol. 5002 of *Lecture Notes in Computer Science*, Springer, p. 7-15, 2007.

Garcés K., Jouault F., Cointe P., Bézivin J., Adaptation of Models to Evolving Metamodels, Technical report, INRIA, 2008.

Girschick M., Difference Detection and Visualization in UML Class Diagrams, Technical report, TU Darmstadt, 2006.

Javed F., Mernik M., Gray J., Bryant B. R., « MARS : A metamodel recovery system using grammar inference », *Information & Software Technology*, vol. 50, n° 9-10, p. 948-968, 2008.

Jouault F., Bézivin J., « KM3 : a DSL for Metamodel Specification », *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*, Bologna, Italy, p. 171-185, 2006a.

Jouault F., Bézivin J., Kurtev I., « TCS : a DSL for the specification of textual concrete syntaxes in model engineering », *in* S. Jarzabek, D. C. Schmidt, T. L. Veldhuizen (eds), *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, Proceedings*, ACM, p. 249-254, 2006b.

Jouault F., Kurtev I., « Transforming Models with ATL », *Proceedings of the Model Transformations in Practice Workshop, MoDELS 2005*, Montego Bay, Jamaica, 2005.

Kolovos D. S., Paige R. F., Rose L. M., Polack F. A., *Epsilon*, `http://epsilonlabs.svn.sourceforge.net/svnroot/epsilonlabs/org.eclipse.epsilon.book/EpsilonBook.pdf`. September, 2008.

Kurtev I., Bézivin J., Jouault F., Valduriez P., « Model-based DSL Frameworks », *Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, OR, USA*, ACM, p. 602-616, 2006.

Melnik S., Garcia-Molina H., Rahm E., « Similarity Flooding : A Versatile Graph Matching Algorithm and ist Application to Schema Matching », *Proc. 18th ICDE*, San Jose, CA, 2002.

Melnik S., Rahm E., Bernstein P., « RONDO - A Programming Platform for Generic Model Management », *Proc. ACM SIGMOD Intl. Conf. Management of Data*, p. 193-204, 2003.

OAEI, *Ontology Alignment Evaluation Initiative*, `http://oaei.ontologymatching.org/`. 2008.

Ohst D., Welle M., Kelter U., « Differences between versions of UML diagrams », *SIGSOFT Softw. Eng. Notes*, vol. 28, n° 5, p. 227-236, 2003.

OMG, *OCL 2.0 Specification, OMG Document formal/2006-05-01*, `http://www.omg.org/docs/ptc/05-06-06.pdf`. 2006.

Rijsbergen C. J. V., *Information Retrieval*, Butterworths, 1979.

Rubin J., Chechik M., Easterbrook S. M., « Declarative approach for model composition », *MiSE '08 : Proceedings of the 2008 international workshop on Models in software engineering*, ACM, New York, NY, USA, p. 7-14, 2008.

Steel J., Jézéquel J.-M., « On model typing », *Software and System Modeling*, vol. 6, n° 4, p. 401-413, 2007.

Treude C., Berlik S., Wenzel S., Kelter U., « Difference computation of large models », *in* I. Crnkovic, A. Bertolino (eds), *ESEC/SIGSOFT FSE*, ACM, p. 295-304, 2007.

Vanhooff B., Ayed D., Baelen S. V., Joosen W., Berbers Y., « UniTI : A Unified Transformation Infrastructure. », *in* G. Engels, B. Opdyke, D. C. Schmidt, F. Weil (eds), *MoDELS*, vol. 4735 of *Lecture Notes in Computer Science*, Springer, p. 31-45, 2007.

Wang T., Pottinger R., « SeMap : a generic mapping construction system », *in* A. Kemper, P. Valduriez, N. Mouaddib, J. Teubner, M. Bouzeghoub, V. Markl, L. Amsaleg, I. Manolescu (eds), *EDBT*, vol. 261 of *ACM International Conference Proceeding Series*, ACM, p. 97-108, 2008.

Xing Z., Stroulia E., « UMLDiff : an algorithm for object-oriented design differencing », *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ACM, New York, NY, USA, p. 54-65, 2005.